第2章 FPGA/ASIC開発の流れを理解する

——LSIの設計でもシステム設計の考えかたが重要

浅井 剛

ここでは，FPGA/ASICがどのような工程を経て開発されているのかについて説明する．昔の回路図による工程と現在のHDLによる工程を比較することで，HDL設計のメリットを明らかにする．実際にやるべき作業をイメージしやすいように，説明にはできるだけ具体的な事例を用いている．　（編集部）

HDL（hardware description language；ハードウェア記述言語）とEDA（electronic design automation；電子設計自動化）ツールによる設計手法の普及により，LSIの設計効率は大きく向上しました．さらに半導体プロセス技術の進歩により，アプリケーションのほぼすべてを一つのLSIで処理するシステムLSI（SOC：system on a chip）を実現できるようになっています．

そこで本稿では，論理設計手法の変遷を，そのベースにある技術とともに説明します．

システム設計とLSI設計

LSIの開発フローを説明する前に，まずそのLSIが組み込まれるシステム全体の開発フローについて考えてみます．**図1**にシステムの開発フローを示します．

● システム設計の流れ

1）仕様設計

仕様設計では，これから開発するシステムの仕様を決定します．

単に製品に組み込む機能だけでなく，形状や目標とする消費電力など，製品カタログの仕様欄に記載されるような内容が含まれます．製品の分野によっては，仕様が決まった時点でカタログを作成して顧客からの反応を集め，必要に応じて仕様へ反映するという手法が採られる場合もあります．

2）機能設計

決まった仕様を受け，それを実現するためのシステム構成を決定します．

まず最初に，表示系，ボタン，スイッチ，コネクタの種類や数など，入出力関連機能を装置の外側から順番に決定します．次にソフトウェアとハードウェアの機能分担をどのようにするかを検討します．

ハードウェアについては，ブロック・レベルの機能分割を行いながら，搭載するマイコンやメモリなどの主要部品，およびコネクタなどの機構部品の選定を行います．一方，ソフトウェアについては，階層を意識して全体の構成を設計し，機能単位に一から開発するのか，もしくは購入品を使うのかを決定します．OSやミドルウェアなどの導入が決まるのもこの機能設計の段階です．

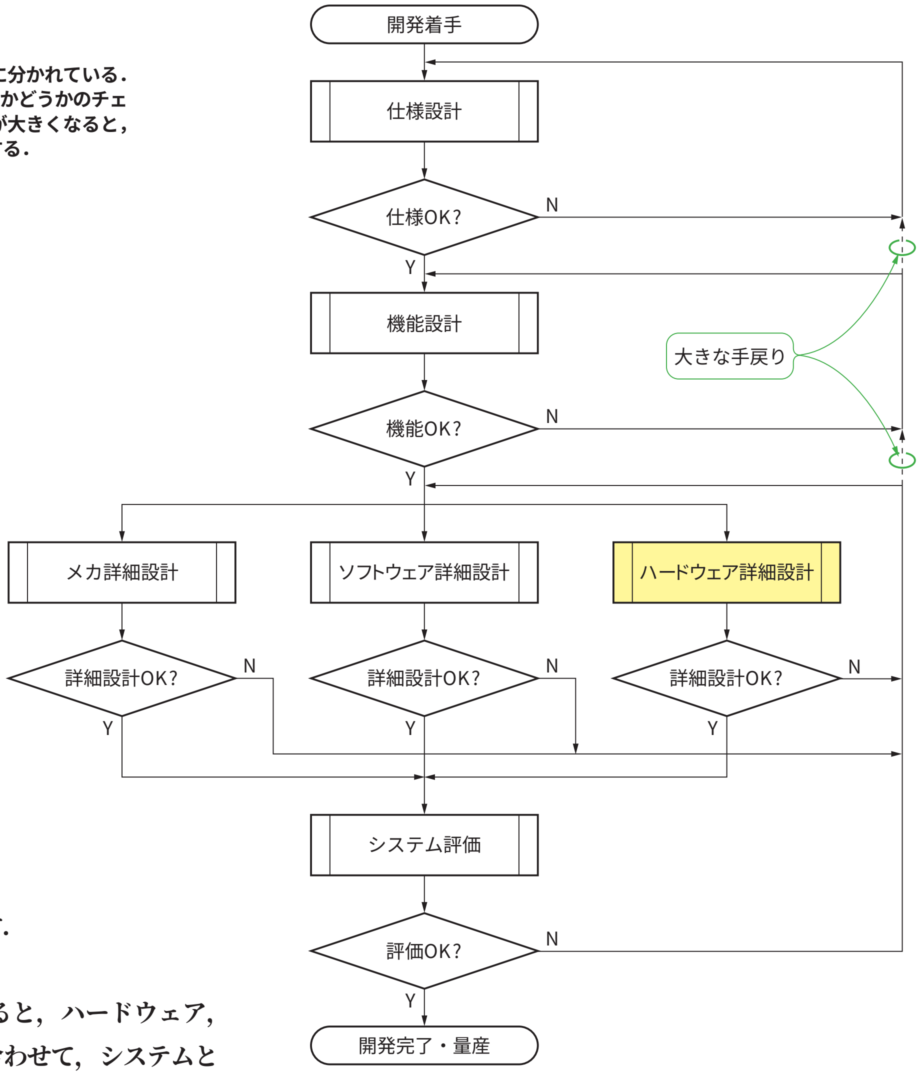
3）詳細設計

与えられた機能をもとに，ハードウェア，ソフトウェア，メカ（機構）のそれぞれについて詳細設計を行います．

ハードウェアは基板レベルの設計が中心で，ASICやFPGAの設計もここに含まれます．ソフトウェアでは，ブート・プログラム，デバイス・ドライバ，アプリケーション・プログラムなどを実際にコーディングしていきます．メカについては，商品の外観となるケース（筐体）の形状など，電子系以外の構成要素を設計していきます．

設計後，ハードウェアについては試作機を組み立て，それが完了するとテスト・プログラムを用いて動作確認を行います．ソフトウェアについてはデバッグ用の記述を随所に盛り込んだプログラムを使用して，できる範囲で単体デバッグを行います．また，メカについても外観の配色，組

図1
システム開発フロー
仕様設計，機能設計，詳細設計，システム評価に分かれている．各ステップの間では，次のステップに進んでよいかどうかのチェックをしっかりと行うことがたいせつ．手戻りが大きくなると，開発期間が延びたり，開発コストが上昇したりする．



み付け，機構類のチェックを行います．

4）システム評価

機能単体による動作確認が終了すると，ハードウェア，ソフトウェア，メカのすべてを組み合わせて，システムとしての評価を行います．

正常動作の機能確認に始まり，ユーザによる誤操作時の保護，ノイズ，消費電力，発熱，動作温度範囲，外観やスイッチ類の操作性にかかわる事項まで，多岐にわたる評価を行います．

● 各工程間のチェックがたいせつ

各工程で次の工程へ進んでよいかどうかをチェックするポイント（マイルストン）がかならずあることに注目してください．チェックでOKとなるまでは次の工程に進めず，前の工程で設計とチェックを繰り返します．

チェックが不十分だと，後工程に入ってから前工程の不ぐあいが見つかることになり，前工程の作業に戻る必要がでてきます．このような大きな手戻りが発生すると，開発期間が延びたり，開発コストが上昇します．最悪の場合，開発した製品の市場競争力がなくなってしまうことさえあります．このチェックがいかに重要かがわかると思います．

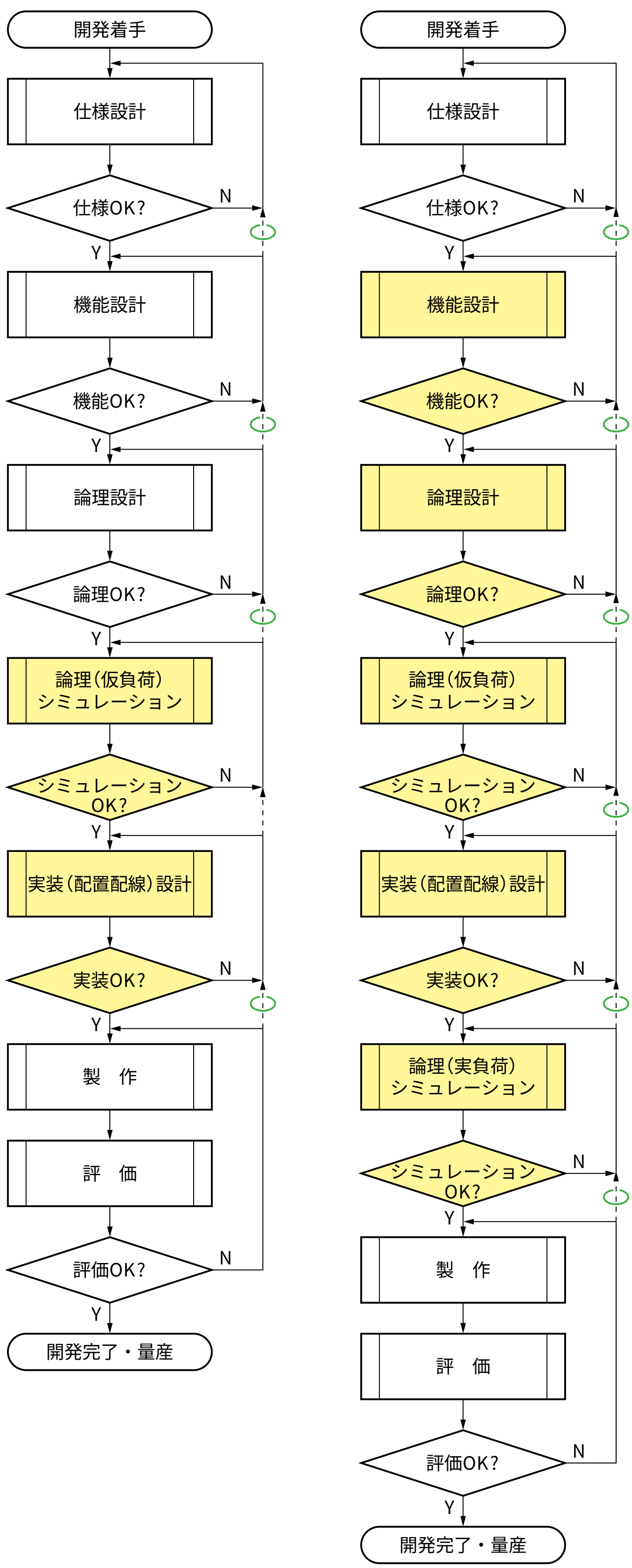
ところで，開発対象が小さいときなどに，すべてのチェックを明確に行わないことがあります．効率化のためにわざと省いているのですが，チェックを甘くしてよいというわけではないので，勘違いしないようにしてください．

● LSI設計もシステム設計とフローは同じ

さて，システム全体の開発フローが理解できたところで，本題のLSI設計フローについて説明します．

図2に，筆者が入社した当時に行っていた回路図ベースのゲート・レベル表現を用いる過去の設計フローと，HDLを使用する現在の設計フローを示します．

仕様設計や機能設計など，図1で示したシステム設計フローと似ていることに気がつかれたでしょうか．設計対象



(a) 回路図ベースの設計　　(b) HDL設計

図2　新旧のLSI設計フロー

図1のシステム設計フローとあまり変わらない．黄色の部分はツールを利用した作業が含まれる工程．HDL設計ではツールによる作業が多いことがわかる．

が違うだけで，一つの製品を開発するというアプローチはまったく同じであるということがわかります．

なお，図2の黄色の部分はツールを利用した作業が含まれる工程です．時代の違いはありますが，HDLへ移行してツールによる作業がいかに増えたかがよくわかります．

回路図によるフィルタ処理LSIの設計

画像処理系でよく使われるフィルタ処理LSIの開発を例に，各設計ステップを説明していきます．

● 仕様設計

システム設計で与えられたLSIに対する機能を実現するための仕様を設計します．データ・シートに記載されるピン配置やレジスタ仕様など，LSIの外部から見える仕様をここで決定します．

1) 取り扱う画像のフォーマット

モノクロ，解像度はVGA(640×480ピクセル)，ノンインターレース，256階調(8ビット)とします．

2) 処理内容

元画像から対象画素の周囲の3×3の領域を切り出して，各画素に対応するフィルタ係数(k)を乗じた後に総和をとり，フィルタ処理後の対象画素の値とします(図3)．

3) 処理速度

処理速度は30フレーム/s(fps：frame per second)とします．これにより，画像のビット・レートはVGA仕様の25.2MHzと決まります．

4) 転送タイミング

画像とともに垂直/水平同期タイミングを外部より入力し，nクロック分の内部処理遅延を経て，結果を外部へ出力します．

5) 設定レジスタ

水平/垂直同期タイミングと有効画像のオフセット・レジスタ，フィルタ計数レジスタなどを設け，マイコンから読み書きできるようにします．

● 機能設計

LSIを機能分割し，ブロック・レベルの設計を行います．各ブロックの構成，機能や動作仕様，ブロック間の接続信号などを決めます．アプリケーションによっては，搭載する処理アルゴリズムの開発が必要になる場合もあります．

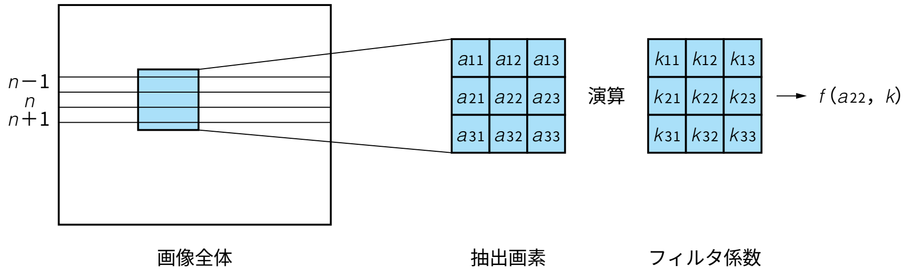


図3
フィルタ処理の概要
元画像から対象画像の周囲の3×3の領域を切り出して，各画素に対応するフィルタ係数（k）を乗じた後に総和をとり，フィルタ処理後の対象画素の値とする．

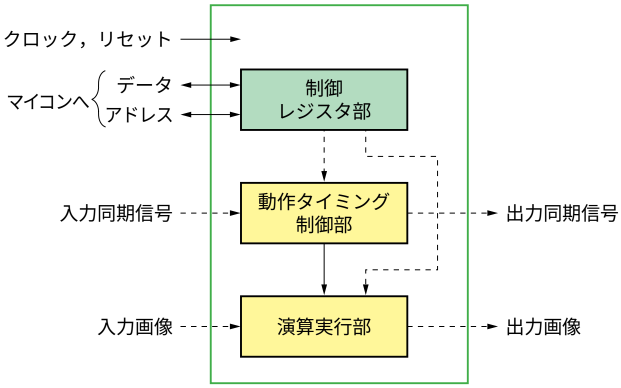


図4　ブロック分割
LSIへ組み込む機能には，それぞれのブロックにおいてある程度共通のものと，ブロックごとに固有のものがある．

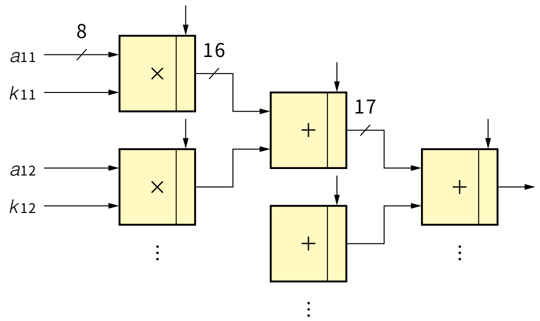


図5　演算実行部の設計
画素と対応するフィルタ係数を乗算し，その後，乗算結果の総和をとる演算を行う．ここでは9個の8ビット乗算器と総和を求める加算器をパイプライン処理のレジスタで区切りながら置いていく．

機能設計のよしあしによって，LSIの性能は大きく変わります．そのため，機能設計を担当するにはそれなりの知識や経験が必要となります．

1）共通部/専用部の切り分けとブロック分割

LSIへ組み込む機能には，それぞれのブロックにおいてある程度共通のものと，ブロックごとに固有のものがあります．

図4のように，クロック，リセット，アドレス/データ・バスは共通の信号です．ブロック間を接続するのは専用の信号になります．また，制御レジスタは各ブロックに共通して存在します．動作タイミング制御と演算実行部は専用の機能です．

1ブロックの機能であっても，規模が非常に大きな場合はサブブロックに分けて，同様の観点で切り分けを行います．取り扱う規模は違っても，各階層における設計スタイルは変わりません．

2）ブロック単位の入出力信号の決定

上述の切り分けによって，各ブロック（サブブロックを含む）へ盛り込む機能が決まったら，そのブロックと周囲を接続する信号の種類，本数，信号のレベルを決定します．

3）仕様との整合性の確認

機能設計完了時点で，仕様設計で決めた処理内容がすべて盛り込まれているかどうかをチェックします．動作条件ごとのレジスタ設定や，パラメータ要素があればその設定範囲も確認します．それ以外にもチェック項目はありますが，この時点のチェックをしっかりやっておくと，仕様変更の原因となる不ぐあいが少なくなり，後々の作業が大幅に楽になります．

● 論理設計

演算実行部を例に説明します．

図5のように，フィルタ演算は，まず画素と対応するフィルタ係数を乗算し，その後，乗算結果の総和をとります．ここでは9個の8ビット乗算器と総和を求める加算器をパイプライン処理のレジスタで区切りながら置いていきます．8ビットどうしの乗算では，演算結果は2倍の16ビットとなります．加算でも1ビット追加になります．ステージの後段になるほどビット数が増加していくことになります．最終的には20ビットとなる総和のMSB（most significant bit；最上位ビット）側8ビットを結果として出力します．

パイプラインの1段に与えられる処理時間とゲートの遅

延時間によっては，複数の加算段を1段にまとめられる可能性もあります．

あとは乗算器と加算器をそれぞれのビット数で設計すればOKです．演算系の回路はビット数に応じてハードウェアの規模は増えるものの，同じ論理が並んでいるだけなので設計の手間はそれほどかかりません．このような論理構成を「規則性が高い」と言います．逆に規則性が低い制御回路などは，一般に「ランダム・ロジック」と呼びます．

ゲート・レベルで設計していた時代は，方眼紙上に手書きで回路図を作成していました．当然のことながら，回路図のチェックや次に説明する論理シミュレーション用のネットリストの作成もすべて人手だったので，多くのケアレス・ミスが発生しました．開発着手時点から完了までに検出されるすべてのバグの約70%が机上で検出可能なレベルであると言われていた時代です．

この経験則はたいへん有効でした．筆者は設計した総ゲート数からバグ件数の予測をたて，実際にバグの検出件数が70%近くに達しないと論理シミュレーションの工程へ移らないなど，手戻り作業を少なくするための指針として使っていました．のちに回路図入力のCADツールが導入され，設計ルール・チェックや論理シミュレーション用のネットリスト生成の工程が自動化され，ケアレス・ミスは大幅に減少しました．

● 論理（仮負荷）シミュレーション

次に，ゲート・レベルのネットリストをもとに，論理シミュレーションによる動作検証を行います．LSIへ実装する前なので，ゲート遅延や配線遅延に標準的な値を用いた仮負荷シミュレーションになります．この段階で扱える論理値は，‘0’，‘1’，‘Z（ハイ・インピーダンス）’，‘X（不定）’程度が一般的です．

ゲート・レベルの回路図をもとに設計していた時代は，シミュレーション・パターン（テスト・ベクタ）もシミュレーションの単位時間に合わせて‘0’，‘1’をすべて記述していくという非常に原始的なものでした（図6）．

当時のシミュレータの動作環境は大型計算機で，結果の出力もライン・プリンタでしたが，自分の設計した論理が思うように動いているのを目で確認できるのが楽しく，時間を忘れて作業に没頭していたことを思い出します．だんだんしごとにも慣れてきて，シミュレーション・パターンを自動生成させたり，シミュレーション結果から動作確認したい箇所を抜き出してチェックしやすくするツールを自作したりと，自分のしごとを効率化するための努力も必要でした．

● 実装設計

ゲートというのはあくまでも論理機能を表現するシンボルで，実際のLSIはMOSFET（metal oxide semiconductor field effect transistor）を組み合わせた回路で動作しています．

図7にインバータを例にした実装設計のイメージを示します．実装設計では，まずゲート回路をトランジスタ回路に変換します．その際，展開したトランジスタの仕様としてゲート幅とゲート長が，接続される負荷の数（fan-out）や動作速度に応じて決定されます．あるいは，あらかじめ設計されていた中から最適な定数を選択します．一般に，プロセス名と呼ばれる0.13 μmや90nmという値はゲート幅を示します．そしてトランジスタ回路を実際のシリコン・ウェハ上に展開します．図7（c）の青色で示した部分が

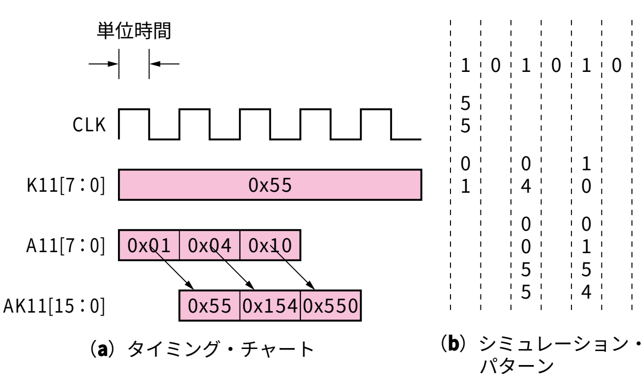


図6　論理シミュレーション

シミュレーションの単位時間に合わせて‘0’，‘1’をすべて記述していく．最後のAK11［15：0］は乗算器からの演算出力である．

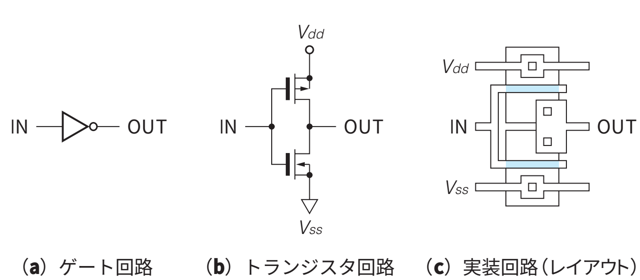


図7　実装設計のイメージ

インバータの例．ゲート回路からトランジスタ回路へ変換し，さらに実際のシリコン・ウェハ上の構造へと変換する．

MOSFETのゲートで，その縦横サイズがゲート長とゲート幅となるように設計します．展開されたゲートをセルと呼ぶこともあります．ゲート回路の構築が完了したら，次にゲート間を配線し，実際にLSIを製造するための各層のマスク・データを作成します．筆者は実装設計については詳しくないので，**図7(c)**はあくまでもイメージとして見てください．

昔はこの実装設計も人手による作業でした．そのチェックは，論理設計者と実装設計者が組になって，お互いが設計した回路の読み合わせを行うというものでした．チップ全体の配線確認では，500～1,000倍程度に拡大した実装図の上を色鉛筆を持った人間がはい回っていました．知らない人からは異様に見えるような作業を，ミスがなくなるまで何度も繰り返すという，非常に単純でかつ根気のいるしごとでした．

● 製作・評価

実装設計が完了してマスクができ上がれば，実際に製造ラインを使ってLSIを製造します．

でき上がったLSIについては，LSIテスタと呼ばれる半導体の試験装置もしくは専用の実装評価ボードを使って，機能や性能などの評価を行います．人手による作業が大半だった時代は，電源とグラウンドが短絡していることもあったので，最初の電源投入時は電流計を見ながら電源電圧を少しずつ上げて，チップの表面に変化がないかどうかをチェックしていたのだそうです．

回路図 vs. HDL

回路図ベースの設計は多くの工数を必要とし，HDL設計と比べるとずいぶん非効率に見えます．しかし，当時は設計ゲート数も少なかったため，論理設計も1人～数人で担当でき，デバイス全体を把握しながら設計できるという貴重な時代でもありました．

HDLが導入されて，設計手法はどのように変わったのでしょうか．回路図ベースの時代と比較しながら説明していきます．

なお，**図2**に示した新旧のLSI設計フローは急に変わったわけではありません．各作業フェーズでEDAツールの導入による効率化の努力はつねに行われていました．HDLによる設計入力に標準化されたことで，EDAツールの開発も促進され，現在に至っているのです．

● 仕様設計

仕様設計は具体的な設計へ入る前なので，回路図ベースの設計とHDL設計における違いはありません．

● 機能設計

ブロック・レベルの設計という意味では，回路図ベースの設計とHDL設計に違いはありません．違うのは設計スタイルです．回路図ベースでは次の論理設計で参照するブロックの仕様書を作成するのに対して，HDL設計では機能設計の段階からHDLで記述し，シミュレーションで機能検証を行います．このレベルで書かれたHDLをビヘイビア(behavior)記述と呼びます．

HDL設計支援ツールが市場に出始めた当初，EDAツール・ベンダのエンジニアはHDL記述そのものを「ドキュメント(仕様書)」と称していました．とくにVHDL(VHSIC HDL)は米国国防総省がVHSIC(Very High Speed Integrated Circuit)の開発を推進するために，論理装置全体の仕様を表現するための言語として考え出されたものなので，なおさらです．設計するブロック(モジュール)ごとに，名まえ，端子名と入出力属性，内部動作が記述されている，まさしく仕様書そのものと考えることができます．

図8を見てください．紙に書いた仕様書からあらためてHDLを記述し直すよりも，最初からHDLで記述したほう

2

図8
仕様書とVHDLによる端子情報の記述
HDL記述そのものが一種のドキュメントである．紙に書いた仕様書からあらためてHDLを記述し直すよりも，最初からHDLで記述したほうが効率的．

端子名	I/O	機能
RESET	I	リセット
CLK	I	クロック
A[7:0]	I	画像入力
K[7:0]	I	係数入力
AK[15:0]	O	乗算出力

(a) 仕様書の記述例

```
entity MULT is
  port(RESET : in  std_logic;
       CLK   : in  std_logic;
        A    : in  std_logic_vector(7 downto 0);
        K    : in  std_logic_vector(7 downto 0);
       AK    : in  std_logic_vector(15 downto 0)
  );
end MULT;
```

(b) VHDLによる記述例

が効率的です．必要ならば該当する部分を引用して仕様書化すればよいのです．また，HDL記述は単純なテキスト・ファイルなので，CVS（Concurrent Versions System）などのバージョン管理システムも適用できます．シミュレーションで検証した記述を次の論理設計でも使用できるため，ファイルの一元化が図れます．

VHDLやVerilog HDLの言語仕様には，非常に抽象度が高くて論理合成ツールがサポートできていないものまで含まれています．一方，シミュレータは言語仕様をほぼ100%サポートしており，抽象度が高い記述でも検証が可能となっています．言い換えると，HDLによる機能シミュレーションは，ゲート展開する前の段階で実施できるということです．さらにHDLによるシミュレーションでは，シミュレーション・パターンや期待値の比較もHDLで記述できます．ハードウェアでありながらソフトウェアと同じようにシンボリック・デバッガが使え，ブレーク・ポイントまで設定することができます．

リスト1にテストベンチの記述例を示します．検証対象と検証環境の関係を記述した，LSI単体よりもう1段上の階層のHDL記述です．クロックを記述していますが，**図6**と比べると楽に記述できることがおわかりいただけるかと思います．

リスト1 テストベンチの記述例

```
entity TB_LSI is
end TB_LSI;

architecture STRUCT of TB
begin
-- LSIのインスタンス
ULSI:LSI port map(RESET, CLK, ･･･)
-- クロック生成
UCLK: process
  begin
    CLK <= ‘1’; wait for 20ns;
    CLK <= ‘0’; wait for 20ns;
  end process;

end STRUCT;
```

このレベルでしっかりシミュレーションを行っておけば，それ以降の工程で機能設計が原因となる不ぐあいが出にくくなり，作業の手戻りが少なくなります．また，機能設計時のシミュレーションで使用したテストベンチ（シミュレーション・パターン）は，以降の仮負荷や実負荷のシミュレーションでも使えます．

● 論理設計

HDLベースの論理設計では，機能設計で記述したブロック名，入出力，およびブロック間接続に関する部分はそのまま使用します．また，ブロック内の動作に関するビヘイビア記述はRTL（register transfer level）記述に書き直し，そのあと論理合成ツールを使ってゲート・レベルのネットリストを生成します．正確な表現ではないかもしれませんが，RTLとは論理合成ツールがゲート・レベルに展開できる範囲のHDL記述ととらえてよいのではないかと思います．

リスト2に演算実行部の記述例を示します．乗算そのものを演算式で記述できるので，非常にすっきりしています．実際には，CLK以外に実行制御部からの制御信号も受けて動作するので，記述はもう少し複雑になります．

回路図ベースの設計との違いは，論理合成ツールがゲート展開を行ってくれるということです．設計者は機能ブロック単位やデバイス全体に関して，動作速度，面積，速度のいずれを優先するのかなどの制約条件を与えてコンパイルし，結果を待つだけです．夜，ツールを実行させて帰宅し，翌朝出社したら，数千～数万ゲートのゲート回路が自動生成されているというのは，ゲートを一つ一つ書いていた時代からみると隔世の感があります．

しかし，このように便利なツールも万能ではありません．シミュレータと論理合成ツールでHDLの文法の解釈が違うことがあり，機能シミュレーションではOKだったのに

リスト2 演算実行部のVHDL記述例

```
architecture RTL of MULT is
signal AK11, AK12, AK13, AK21, AK22, AK23, AK31, AK32, AK33 : std_logic_vector(15 downto 0);
begin
-- 1st stage (MULT)
  process(CLK, A11, A12, ･･･)
  begin
  if CLK’event and CLK = ‘1’ then
    AK11 <= A11 * K11; AK12 <= A12 * K12; AK13 <= A13 * K13;
    AK21 <= A21 * K21; AK22 <= A22 * K22; AK23 <= A23 * K23;
    AK31 <= A31 * K31; AK32 <= A32 * K32; AK33 <= A33 * K33;
  end if;
-- 2nd stage(1st stage of sum)
    ⋮
end RTL;
```

論理合成結果はNGというようなこともありました．設計者が予期しない結果が出てくるのですから，普通であれば「使いものにならない」と判断するところですが，論理合成ツールの圧倒的なパフォーマンスの魅力には勝てませんでした．現在では，どのツールでも同じように解釈する記述を見つけ出し，HDLの記述をその範囲に限定して使うようになりました．

● 論理（仮負荷）シミュレーション

論理合成ツールがそれなりに最適化した論理のネットリストを用いて，回路図ベースの場合と同じように仮負荷によるシミュレーションを行います．

機能シミュレーション時のテストベンチをそのまま使うことができるうえ，結果の比較も可能なので，機能設計段階との整合性の確認も容易です．

ゲート・レベルのシミュレーションであるため，マクロ化された機能設計時のシミュレーションと比べると，処理速度は大幅に低下します．

● 実装設計

HDL設計とは無関係ですが，LSIの高集積化に対応するため，実装設計ツールも時代とともに発展してきました．セルの自動配置，ゲート間の自動配線，レイアウト・ルールのチェック，およびゲート・レベルのネットリストとの一致確認ツールなどが整備され，人手に頼っていたら絶対に実現できないような大規模LSIの開発が可能となりました．

● 論理（実負荷）シミュレーション

完了した実装ファイル（レイアウト情報）から，すべてのゲートおよび配線に関する遅延情報を算出します．実負荷シミュレーションでは，その遅延情報を含め，より実物に近い条件でシミュレーションを行います．

ここでも仮負荷と同じテストベンチが使え，結果の照合もできるので，運が良ければ全パターンを一気に実行して，確認終了なんてこともあるかもしれません．

● 製作・評価

技術の進化によって製作・評価のステップもずいぶん効率化されていますが，作業内容に大きな変化はありません．ただ，LSIの大規模化によって内蔵する機能の数や複雑さなどが増したため，実機による機能確認により多くの時間が必要になっています．また，動作速度も非常に高速になり，従来であればユニバーサル基板＋ジャンパ配線でよかったものが，きちんとしたプリント基板を製造してからでないと評価できない場合も出てきています．

システムの中で使いやすいLSIとは

「使いかたを知らない職人が作った道具は使いにくいものにしかならない．実際に自分で使ってみるだけでなく，多くの使い手の声に耳を傾け，その声を取り入れた職人だけが真に使いやすい道具を作り出せる」という話を聞いたことがあります．筆者はこのことばがすべての製品作りに当てはまると信じています．

マイコンに対する割り込み機能を持ったLSIを例に，システム側から見たLSI設計の考えかたを示します．

● 要求概要

開発するLSIの割り込み制御部に対する要求概要が下記の3点だとします．

1）マイコンに対して割り込みをかけられること．割り込み出力は1本とする．マイコンが割り込み要因を識別完了するまで，割り込み出力は保持されること．

2）割り込み要因は全部で3本ある．個別に割り込みのイネーブル/ディセーブル制御ができること．

3）割り込みハンドラでどの割り込みが発生したのかを識別できるしかけを持つこと．

マイコンには8ビットの外部データ・バスがあるものとして，具体的な外部仕様を設計していきます．

● 外部仕様の決定

マイコンがアクセスするレジスタと，設定や割り込み入力に対する動作を決定します．ここで検討しなければならない項目は多岐にわたっています．

1）レジスタの数

要求概要2）から割り込みイネーブル設定用に3ビット，要求概要3）から割り込み要因を識別するためにも3ビット，の合計6ビットのレジスタがあればよいことがわかります．そこで，この6ビットをどのようにレジスタに割り付けたらよいのかを考えてみましょう．

まず，考えられる割り当てかたをできる限り列挙します．例を**表1**に示します．6ビット分の割り付けかたですが，無

2

表1　レジスタ割り付け仕様の比較

仕様	メリット	デメリット
1レジスタ	すべてのアクセスを1アドレスでできる	仕様変更でビット数が増えたときなど，将来の拡張性が低い
2レジスタ（読み出し用と書き込み用）	アクセス・メソッドで分類されているので，制御ソフトウェアが組みやすい	将来の拡張性を考えすぎで，レジスタ中の空きビットが多い
3レジスタ（割り込み要因ごとに1レジスタ）	機能単位が分かれているので，制御ソフトウェアが組みやすい	割り込み要因をチェックするのに毎回三つのレジスタをアクセスする必要がある
6レジスタ（1レジスタ/1ビット）	ほかのビットの設定を意識しないですむ	占有アドレス空間が多い．制御ソフトウェアが煩雑になる

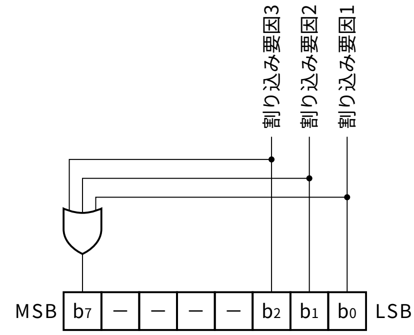


図9　マイコンの機能を利用したビット追加例

LSB側の3ビットに割り込み要因を割り付け，いずれの割り込みが発生してもMSBのビットをセットする機能を追加している．このレジスタを読み込むと，Nフラグの変化により割り込み発生の有無がわかる．

理矢理考えれば，1レジスタから6レジスタまであります．そのすべてを列挙していたのではきりがないので，最初の段階で常識的な範囲に絞り，それぞれのメリットとデメリットを検討します．

最適解というものは存在せず，選択条件とともに変化します．今回は制御ソフトウェアの組みやすさを優先し，2レジスタもしくは3レジスタに絞り込み，さらに割り込み発生時の処理時間が短い2レジスタ案を採用することとします．

2）レジスタ中のビット割り付け方法

書き込み用レジスタと読み出し用レジスタに分けることが決まったら，その8ビット・データのどこにビットを割り付けるかを考えます．割り込み要因に優先度があれば，その順にLSB（least significant bit；最下位ビット）詰めもしくはMSB詰めというのが一般的です．また，接続されるマイコンによっては，仕様にひとくふうするだけでもっと使いやすくなるはずです．

マイコンには，コンディション・コード・レジスタ（CCR：condition code register）があります．マイコンによっては名称が異なることもありますが，条件分岐などのために用いられるもので，演算結果によってビットが変化するのが普通です．しかし，汎用レジスタへのデータ転送命令でもそのデータに応じてCCRをセットするマイコンがあります．例えば，米国Freescale Semiconductor社（旧Motorola社）の8ビット・マイコンのCCRはこの仕様を備えています．

CCRには演算結果の正負を示すN（Negative）フラグがあります．8ビット・データの場合なら2の補数で－128～＋127までの値を表現できますが，このときMSBの状態がそのままNフラグの値となります．このしくみを利用することを考えてみましょう．

図9を見てください．I/OポートのLSB側の3ビットに割り込み要因を割り付け，いずれの割り込みが発生してもMSBのビットをセットする機能を追加しています．これにより，割り込みを使わない場合でも，このポートを読み出してデータが汎用レジスタに書き込まれると，CCRのNフラグがレジスタ値の影響を受けることになります．つまり，読み出したデータに対してビットを確認しなくても，割り込みの有無を判断できることになります．

C言語によるソフトウェア開発が主流となった今，コンパイラがCPUの機能を十分に理解していても，ここまで意識したコードを生成できることはほとんどありません．設計者がCPUのしくみを理解し，場合によってはビルトイン・アセンブラを使って記述しないかぎり実現できないでしょう．

設計者の自己満足にすぎないと思われるかもしれません．しかし，だれが使っても使いやすいと言われるLSIは，このような細かいくふうの積み重ねで開発されているのです．

参考・引用*文献

（1）萩原宏；電子計算機通論1，朝倉書店，1969年11月．
（2）R. Lipsett，C. Schaefer，C. Ussery著，杉山尚志監訳；VHDL言語設計によるハードウェア設計へのアプローチ，マグロウヒル出版，1990年6月．

あさい・たけし
日立エンジニアリング（株）

<筆者プロフィール>

浅井剛．20数年前に入社．専用/汎用プロセッサの開発を本業としながら，アプリケーション・ボードのハード/ソフト開発に従事．その後，マイコン＋ASIC/FPGAを搭載した組み込みシステムの開発を担当．現在は組み込みシステム開発のかたわら，FPGAによるSOCに関する執筆や講演を行っている．